**SCANDAL:** Static Analyzer for Detecting Privacy Leaks in Android Applications

Jinyung Kim, Yongho Yoon, and Kwangkeun Yi  
Programming Research Laboratory  
Seoul National University  
Seoul, Korea  
{ jykim, yhyoon, kwang } @ ropas. snu. ac. kr

Junbum Shin  
SW R&D Center  
Samsung Electronics  
Suwon, Korea  
junbum. shin@samsung.com

**Abstract**—Smartphone applications can steal users’ private data and send it out behind their back. The worldwide Android smartphone market is growing, which raises security and privacy concerns. However, current Android’s permission-based approach is not enough to ensure the security of private information.

In this paper, we present SCANDAL, a sound and automatic static analyzer for detecting privacy leaks in Android applications. We analyzed 90 popular applications using SCANDAL from Android Market and detected privacy leaks in 11 applications. We also analyzed 8 known malicious applications from third-party markets and detected privacy leaks in all 8 applications.

**Keywords**—Android; privacy; static analysis; security;

**I. INTRODUCTION**

**A. Problem**

Smartphone applications can steal users’ private data and send it out behind their back [6], [7]. Smartphones store various personal data, such as phone identifiers, location information, and contacts. Third-party applications, which can be downloaded freely at markets, frequently access the data. Most of the applications do so to explore the fun and utility of smartphone technology. However, such accesses also raise concerns and issues of privacy risk.

Android’s permission-based approach is not enough to ensure the security of private information. [3], [9]. Android requires application developers to declare the permissions so their applications can access users’ private information. However, the permissions do not let you know the actual trace of private data. It is uncertain if an application only accesses private data locally, or sends the data out. Also, developers tend to request more permissions than what they need [10]. As a result, users also tend to care less about the permissions when they install applications.

**B. Our Solution**

We developed a static analyzer SCANDAL that detects privacy leaks in Android applications. SCANDAL determines if there exists any flow of data from an information source through a sink. SCANDAL is a sound analyzer. It covers all possible states which may occur when using the application.

In other words, SCANDAL can detect every possible privacy leak in the application.

The following is a simple example of an interprocedural privacy leak SCANDAL detected. The example code is from the Dalvik bytecode of Google Wallpaper 4.2.2. This application sends a device ID, called IMEI, to the content server. At line 2, the application gets device ID by calling the getDeviceId API and stores it in a global variable. After that, in the getLocale_version_IMEI_W_H method, the IMEI is loaded and is appended to some other string values and returned. The returned string is passed to the getSearchURL method, and also manipulated and returned to initTagWebView. Finally, the string that contains IMEI is made into a URL and sent to the content server of the application.
The following is another example of a privacy leak SCANDAL detected. The example code is from the Dalvik bytecode of Bible Quotes. At line 3, this application sends location information to an advertisement server. The application obtains the location information by calling the `getLatitude` API and stores it in an array. After that, the information passes various manipulation steps and is made as a part of an HTML code, which renders the advertisement. Finally, the advertisement is shown on the WebView.

```
protoFromLocation()
    new-array r1, r1, Object
    callv Location.getLatitude(r10)
    move-result r3
    mul-double r3, r7
    double-to-long r3, r3
    calld Long.valueOf(r3)
    move-result r3
    aput-object r3, r1, r2
    calld String.format(r0, r1)
    move-result r0
    return r0

getLocationParam()
    callv protoFromLocation(r6, r5)
    move-result r1
    const-string r5 "e1+"
    callv StringBuilder.append(r0, r5)
    callv StringBuilder.append(r0, r1)
    ...
    callv StringBuilder.toString(r0)
    move-result r5
    return r5

generateHtml()
    callv getLocationParam(r11)
    move-result r4
    calld AdSpec.Parameter.<init>(r11, r12, r4)
    callv List.add(r5, r11)
    ...
    new r11, StringBuilder
    calld StringBuilder.<init>(r11)
    const-string r12,
    ";\n    callv StringBuilder.append(r11, r12)
    move-result r11
    callv AdSpec.getAdUrl(r15)
    move-result r12
    callv StringBuilder.append(r11, r12)
    move-result r11
    const-string r12,
    "">
    callv WebView.loadData(r1, r0, r2, r3)

A brief video demo of the experiments is available at our website [16].

C. Organization

Sections 2 through 4 describe an overview of SCANDAL, Section 5 shows the experiment results, Section 6 discusses the limitations, Section 7 describes related work, and Section 8 concludes.

II. ANALYSIS TARGET

We build a static analysis framework for Android programs which takes Dalvik VM bytecode as an input to detect privacy leaks in Android applications.

A. Target: Privacy Leaks

SCANDAL determines if there exists a flow of data from an information source through a sink. We call such a flow a privacy leak.

1) Sources: API calls that return private information are considered as information sources. We track 3 types of private information:

- **Location Information**: Applications can send users’ physical location to remote advertisement servers [7].
- **Phone Identifiers**: Applications can send phone identifiers to remote network servers [6], [7]. We track four phone identifiers: phone number, IMEI (device ID), IMSI (subscriber ID), and ICC-ID (SIM card serial number).
- **Eavesdropping**: We track both audio (on microphone) and video (on camera) eavesdropping.

2) Sinks: API calls that can transfer data to the network, file or SMS are considered as sinks.

- **Network/File**: Relevant API calls in the WebView, OutputStream, DataOutputStream, and HttpURLConnection objects are considered as sinks.
- **SMS**: `sendTextMessage()` function in object SmsManager is also considered as a sink.

Whenever private data is sent out of the phone, SCANDAL considers it as a privacy leak. Such a leak might reasonably
be expected by users, depending on the functionality of the application. We do not try to understand the intention of the situations.

B. Target Language: Dalvik VM Bytecode

Dalvik is the register-based virtual machine in the Android operating system. Android developers commonly write an application in a dialect of Java, and the application is compiled to bytecode.

SCANDAL deals with the Dalvik bytecode, rather than translating the Dalvik bytecode to Java. We do this because known reverse engineering tools, such as dex2jar, fail in some cases. It is also possible for malicious developers to deliberately modify an application at the bytecode level, which makes the application hard to be decompiled. By dealing directly with the bytecode, we do not have to suffer from such issues.

III. HOW SCANDAL WORKS

In this section, we discuss the architecture of SCANDAL and how SCANDAL works.

We designed SCANDAL in the abstract interpretation framework [4], [5]. We designed Dalvik Core, an intermediate language for the analyzer. We defined the concrete and abstract semantics of Dalvik Core. Our analysis is path-insensitive, context-sensitive with a context depth 1, and adopts a hybrid of flow-sensitive and flow-insensitive analysis.

A. Front-end

SCANDAL takes a packaged Android application (.apk) file as an input. We can extract the bytecode of the application as a Dalvik Executable (.dex) file from the package.

Front-end consists of dumping and parsing of the Dalvik Executable file. For the dump part, we modified the existing reverse assembler DexDump. DexDump is included in the Android SDK [2]. The original DexDump is made for human readability, so the dumped result is not easy to be parsed directly. We modified the code of DexDump so that the dumped result could be parsed easily. Moreover, the original DexDump drops some key information such as switch tables or initial data of arrays. We also modified the code so it does not omit any important data.

B. Dalvik Core

We designed Dalvik Core, an intermediate language, for simple and efficient analysis. There are over 220 Dalvik instructions, but a lot of the instructions have semantics similar to some other instruction but differ only in detail. For example, there are more than 20 instructions that move a value into a register. The intermediate language consists of 15 instructions and can represent all of the original Dalvik instructions.

We define the syntax of Dalvik Core in Figure 2. A program is a tuple of instructions, method table, handler table, and subtype relation. Each instruction is labeled with the program counter, in an increasing order from 0. Method table stores entry points of the methods. Handler table stores exception handling stacks for every program point. The symbol * and ◦ indicate various operators.

We define the concrete semantics of Dalvik Core in Figure 3. Collecting semantics of a program, which our analysis will safely approximate, is the set of all machine states occurring during the execution of the program for all inputs. This set is defined as the least fixed point of

$$\lambda S. S_0 \sqcup \text{Next } S$$

where $S_0$ is the set of all initial states and the $\text{Next}$ function does the one-step execution by applying the transition operation $\rightarrow$ to each state in $T$.

A machine state is a tuple of memory, environment, callback environment, program counter, and continuation. Memory models the heap memory while environment is for registers. Callback environment contains event-handlers which have been registered at the state. Continuation captures return points of the methods. State transition operation $\rightarrow$ is defined for every instructions. Function $\nu$ defines the value of an expression under its environment. We used usual record notation for record values. We considered an array value as a record, but with an integer index as a field name. $lc$ indicates a reserved location for global variables. $\text{Handler}$ is a function that returns the exception handler which should be used to handle a given exception.

Our analyzer is sound with respect to Dalvik Core; anything that is not represented in the intermediate language could lead to unsoundness. For example, using Android APIs which are not manually encoded in the analyzer might lead to an unsound result. It is possible to design the analyzer sound even in those cases, but then the precision (true alarm ratio) will degrade drastically.

C. Translation

We designed and implemented a translator to convert Dalvik VM bytecode into Dalvik Core. The translator builds
control flow graph, adds hard-coded definitions of library calls, and create the main entry point according to our model of the whole execution of a Dalvik application.

Figure 1 describes the execution model of a Dalvik application. We divided an execution into two phases: initializing phase and event-handling phase. During the initializing phase, the application constructs the activity objects and initializes static variables. After initialized, the application waits for events, and calls the appropriate callback methods, called listeners, when an event occurs.

**D. Abstract Interpretation**

SCANDAL is an abstract interpreter. To detect all potential privacy leaks in Android applications, SCANDAL considers every machine state which may occur when executing applications. It computes a sound approximation of every machine state of the Dalvik Core program.

We define the abstract semantics of Dalvik Core in Figure 4. The abstract semantics follows conventional abstract interpretation designs. Abstract machine states are partitioned into sets of states by program point. The abstract semantics is the least fixed point of a function that transfers an abstract trace into next abstract trace. State transition operation \( \rightarrow \) is defined for every instruction. Formally, our analyzer computes a fixed point of the following abstract semantic function

\[
\lambda \hat{S}. \hat{S}_0 \sqcup \hat{N}ext\hat{S}
\]

which is proven, within the abstract interpretation framework, a safe approximation of the collecting semantics (1). The \( \hat{S}_0 \) is a table from program points to their abstract initial states, and the \( \hat{N}ext \) function approximates the one-step execution by applying the abstract transitions operation \( \rightarrow \).

Most of the abstract domains are also defined conventionally. Integer domain is defined as the flat domain. Heap memory locations are abstracted by sets of allocation sites. All the abstract values of elements in an array are joined together. String values have a special domain called Prefix Domain, which keeps prefixes of strings. For example, in Google Wallpaper 4.2.2 we introduced in section 1.B, the leaked string which contains the device ID is represented as “http://www.imnet.us/api/wallpapers/photos/search_keywords?”*. A device ID follows the constant part of the string. Since the device ID is unique for each phone, it cannot be specified during static analysis. But SCANDAL can analyze prefixes of a string, which in this case helps identify the sink of the leak. Other values have power set
\[
\text{Concrete Semantics}
\]

\[
\begin{align*}
\text{Trace} & = \text{State}^+ \\
\text{st} & \in \text{State} = \text{Mem} \times \text{Env} \times \text{CBEnv} \times \text{PC} \times \text{Conti} \\
M & \in \text{Mem} = \text{Loc}^{\text{in}} \rightarrow \text{Obj} \\
\sigma & \in \text{Env} = \text{Reg}^{\text{in}} \rightarrow \text{Val} \\
C & \in \text{CBEnv} = \text{2}^{\text{id} \times \text{Val}} \\
K & \in \text{Conti} = (\text{Env} \times \text{PC})^* \\
l & \in \text{Loc} \\
r & \in \text{Reg} \\
\text{obj} & \in \text{Obj} = \text{Type} \times (\text{Id}^{\text{in}} \rightarrow \text{Val}) \\
v & \in \text{Val} \\
v & ::= \text{if} \mid \text{let} \mid \text{str} \mid \text{ty} \\
\end{align*}
\]

\[
\langle M, \sigma, C, pc : \text{move} \rangle, K \rangle \rightarrow \langle M, \sigma \{ r \mapsto \forall \sigma \ E \}, C, pc + 1, K \rangle
\]

\[
\langle M, \sigma, C, pc : \text{istype } r_d r_t, K \rangle \rightarrow \langle M, \sigma \{ r_d \mapsto 1 \}, C, pc + 1, K \rangle
\]

\[
M(\sigma(r)) \text{.ty} \leq \text{ty}
\]

\[
\langle M, \sigma, C, pc : \text{istype } r_d r_t, K \rangle \rightarrow \langle M, \sigma \{ r_d \mapsto 0 \}, C, pc + 1, K \rangle
\]

\[
M(\sigma(r)) \text{.ty} \not\leq \text{ty}
\]

\[
\langle M, \sigma, C, pc : \text{new } r_d ty, K \rangle \rightarrow \langle M, \{ l \mapsto \{ \}\} \sigma \{ r_d \mapsto l \}, C, pc + 1, K \rangle
\]

\[
l \notin \text{Dom}(M)
\]

\[
\langle M, \sigma, C, pc : \text{get } r_d r_o id, K \rangle \rightarrow \langle M, \sigma \{ r_d \mapsto M(\sigma(r_o)).id \} \}, C, pc + 1, K \rangle
\]

\[
\langle M, \sigma, C, pc : \text{put } r_s r_o id, K \rangle \rightarrow \langle M \sigma(r_o) \mapsto M(l) + \{ \text{id} = \sigma(r_s) \} \}, C, pc + 1, K \rangle
\]

\[
\langle M, \sigma, C, pc : \text{get } r_d id, K \rangle \rightarrow \langle M, \sigma \{ r_d \mapsto M(l) + \{ \text{id} = \sigma(r_s) \} \}, C, pc + 1, K \rangle
\]

\[
\langle M, \sigma, C, pc : \text{addcallback } id r_a, K \rangle \rightarrow \langle M, \sigma, C \cup \{ (\text{id}, \sigma(r_a)) \}, pc + 1, K \rangle
\]

\[
\langle M, \sigma, C, pc : \text{call } ty id r_a r_b ..., K \rangle \rightarrow \langle M, \{ r_o \mapsto \sigma(r_a), r_1 \mapsto \sigma(r_b), ... \}, C, MT(\text{ty}, id), (\sigma, pc); K \rangle
\]

\[
\langle M, \sigma, C, pc : \text{vcall } id r_a r_b ..., K \rangle \rightarrow \langle M, \{ r_o \mapsto \sigma(r_a), r_1 \mapsto \sigma(r_b), ... \}, C, MT(M(\sigma(r_a)).ty, id), (\sigma, pc); K \rangle
\]

\[
\langle M, \sigma, C, pc : \text{return } (\sigma', pc'); K \rangle \rightarrow \langle M, \sigma', C, pc' + 1, K \rangle
\]

\[
(\sigma', pc', K') = \text{Handler}(\sigma, pc, K, M(\sigma(r)).ty)
\]

\[
\langle M, \sigma, C, pc : \text{throw } r, K \rangle \rightarrow \langle M, \sigma' \{ r_{ex} \mapsto \sigma(r) \}, C, pc', K' \rangle
\]

\[
\langle M, \sigma, C, pc : \text{jmpnz } e pc', K \rangle \rightarrow \langle M, \sigma, C, pc', K \rangle
\]

\[
\forall \sigma e \neq 0
\]

\[
\langle M, \sigma, C, pc : \text{jmpnz } e \_ , K \rangle \rightarrow \langle M, \sigma, C, pc + 1, K \rangle
\]

\[
\forall \sigma e = 0
\]

\[
\langle M, \sigma, C, pc : \text{switch } e \_ ST, K \rangle \rightarrow \langle M, \sigma, C, pc', K \rangle
\]

\[
(V \sigma e, pc') \in ST
\]

\[
\langle M, \sigma, C, pc : \text{switch } e \_ ST, K \rangle \rightarrow \langle M, \sigma, C, pc', K \rangle
\]

\[
(V \sigma e, _) \notin ST
\]

\[
\langle M, \sigma, C, pc : \text{wait}, K \rangle \rightarrow \langle M, \{ r_0 \mapsto l, r_1 \mapsto \text{input()}, ... \}, C, MT(M(l).ty, id), (\sigma, pc); K \rangle
\]

\[
(id, l) \in C
\]

Figure 3: Concrete Semantics
\[\langle M, \hat{\sigma}, \hat{C}, pc : \text{move} \ r, \ e, K \rangle \stackrel{\mu}{\rightarrow} \langle M, \hat{\sigma} \{ r \mapsto \hat{\sigma} \ e \}, \hat{C}, pc + 1, K \rangle\]

\[\forall l \in \hat{\sigma}(r_i) \forall ty' \in \text{Dom}(\hat{M}(l)) \ (ty' \neq ty) \]

\[\langle M, \hat{\sigma}, \hat{C}, pc : \text{vcall} \ id \ r_a \ r_b \ldots, K \rangle \stackrel{\mu}{\rightarrow} \langle M, \{ r_0 \mapsto \hat{\sigma}(r_a) \ldots \}, \hat{C}, MT(ty, id), \{ pc \} \rangle \quad l \in \hat{\sigma}(r_i), ty \in \text{Dom}(\hat{M}(l))\]

\[\end{align*}\

\[\text{Figure 4: Abstract Semantics}\]
domains.

To detect privacy leak, we collect information on where the value was created. When a value is created at an information source, we denote the program counter of the source and send it through the analysis. When a value is created from existing values, we denote the union of all sets of the program counters from existing values. By this, we can collect every values which could be created from information sources. If such values flow out through an information sink, SCANDAL detects it and consider it as a privacy leak.

SCANDAL runs both flow-sensitive and flow-insensitive analysis for different part of an application. For the initialize phase, SCANDAL runs flow-sensitive analysis for better precision. On the other hand, during the event-handling phase, the order of executions of methods can be random. So during the second phase, flow-sensitive analysis does not give us enough precision despite of analysis cost. Therefore, for the second phase, SCANDAL runs flow-insensitive analysis.

IV. ANALYSIS OF DALVIK’ S CHALLENGING FEATURES

A. Library Call

The analyzer has to know the semantics of library function calls to analyze precisely. There are about 3,000 API classes in the Android platform, and each class contains from several to dozens of methods. The source code of such classes and methods are not included in the application.

To maintain the precision, SCANDAL includes the definitions of popular API methods and classes. We chose 220 methods which are frequently used in the popular applications we collected for the experiments. The semantics of those methods are encoded in our core language. The front-end translator transforms the library method calls into the encoded core language instruction sequences.

B. Implicit Method Invocation

The analyzer should handle methods that are never explicitly invoked in the program. Such methods might be dead code, but it is also possible that they are intended to be invoked implicitly. Every Android application frequently includes methods that are not explicitly invoked, such as Listeners, Threads, and Intents.

SCANDAL handles implicit method invocation. We manually coded the semantics and calling conventions of the popular API functions which add listeners, initialize threads, or activate an activity by an intent. For example, an instruction that calls Button.setOnClickListener() is translated into a command of Dalvik Core as addcallback Button.onClickLIsten.

C. Reflection

Reflection is a feature of Java that allows dynamic code generation. It is possible to instantiate new objects and invoke methods from the name of the classes or methods.

SCANDAL handles simple cases of reflection by analyzing string values. We run a string analysis using prefix domain, which keeps a prefix of a string as the abstracted value. By this string analysis, we can narrow down the names of objects that should be instantiated. For example, SCANDAL handles method Class.forName() by creating java.lang.Class objects according to the abstract string value of the argument.

V. EXPERIMENTS

In this section, we show experiment results of SCANDAL. We tested our analyzer with applications from the official market and third-party markets.

A. Official Android Market Applications

We analyzed 90 free applications from Android Market [1]. 10 popular free applications from 9 random categories are chosen, as in July 2011. We extracted packaged applications as .apk files using adb (Android Debug Bridge), which is included in the Android SDK [2].

SCANDAL detected privacy leaks in 11 applications. Table 1 summarizes the analysis result. We manually inspected the result and confirmed every application included actual leaks. We also identified the sinks.

1) Location to advertisement server: 6 applications sent location information to advertisement servers. 3 applications included AdMob modules and the other 3 included AdSenseSpec modules. Location information was either sent directly to their servers, or used as the part of the HTML code which WebView module loads to render the advertisement.

2) Location to file or analytics tool server: 5 applications outputted location information as a file, and some of the applications are believed to send location information to an application analytics tool server (http://data.flurry.com/aar.do). All 5 applications included FlurryAgent module in their bytecode.

3) Phone identifiers to remote server: 1 application sent out IMEI to a remote server.

4) Eavesdropping: We found no applications that have video or audio eavesdropping behaviors.

<table>
<thead>
<tr>
<th>Application</th>
<th>size</th>
<th>time</th>
<th>mem</th>
<th>Detected leak</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kids Preschool Puzzle</td>
<td>87</td>
<td>1</td>
<td>36</td>
<td>Location*</td>
</tr>
<tr>
<td>Job Search</td>
<td>167</td>
<td>1</td>
<td>108</td>
<td>Location**</td>
</tr>
<tr>
<td>Kids Shapes</td>
<td>225</td>
<td>1</td>
<td>155</td>
<td>Location*</td>
</tr>
<tr>
<td>Kids ABC Phonics</td>
<td>134</td>
<td>3</td>
<td>119</td>
<td>Location*</td>
</tr>
<tr>
<td>Backgrounds HD Wallpapers</td>
<td>109</td>
<td>4</td>
<td>141</td>
<td>IMEI</td>
</tr>
<tr>
<td>Bible Quotes</td>
<td>138</td>
<td>8</td>
<td>263</td>
<td>Location**</td>
</tr>
<tr>
<td>ES Task Manager</td>
<td>158</td>
<td>9</td>
<td>423</td>
<td>Location**</td>
</tr>
<tr>
<td>Multi Touch Paint</td>
<td>198</td>
<td>47</td>
<td>727</td>
<td>Location**</td>
</tr>
<tr>
<td>A dao File Manager</td>
<td>255</td>
<td>62</td>
<td>1149</td>
<td>Location**</td>
</tr>
<tr>
<td>(D-Day) The Day Before</td>
<td>293</td>
<td>224</td>
<td>2657</td>
<td>Location**</td>
</tr>
<tr>
<td>Kids Numbers and Math</td>
<td>101</td>
<td>538</td>
<td>185</td>
<td>Location*</td>
</tr>
</tbody>
</table>

Table I

PRIVACY LEAKS DETECTED IN ANDROID MARKET APPLICATIONS
size is the size of the dex (Dalvik Executable) file (KB).
time is the CPU time spent (sec). mem is the peak memory
consumption (MB). Flurry* and advertisement** servers are
identified.

B. Black Market Applications

We analyzed 8 known malicious applications from third-
party (“black”) markets. These applications are originally
free and can be downloaded via Android Market. However,
infect ed applications, which seems to be the same as original
ones, can be found in third-party markets. We downloaded
.apk files of these applications from the Internet.

Table 2 summarizes the analysis result. All of the 8
applications sent out the phone number, IMEI, IMSI, ICC-
ID, and the location information. We manually inspected the
result and confirmed every application included actual leaks.
When the infected applications are executed, it immediately
sends several private information to a remote host, which is
believed to be a malicious server.

<table>
<thead>
<tr>
<th>Application</th>
<th>size</th>
<th>time</th>
<th>mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shot Gun</td>
<td>95</td>
<td>36</td>
<td>164</td>
</tr>
<tr>
<td>Baseball Superstars 2010</td>
<td>165</td>
<td>61</td>
<td>285</td>
</tr>
<tr>
<td>CacheMate for Root Users</td>
<td>174</td>
<td>67</td>
<td>245</td>
</tr>
<tr>
<td>Monkey Jump 2</td>
<td>169</td>
<td>74</td>
<td>442</td>
</tr>
<tr>
<td>Protector</td>
<td>107</td>
<td>75</td>
<td>209</td>
</tr>
<tr>
<td>Gold Miner</td>
<td>191</td>
<td>81</td>
<td>481</td>
</tr>
<tr>
<td>Mini Army</td>
<td>480</td>
<td>174</td>
<td>1922</td>
</tr>
<tr>
<td>Xing Metro</td>
<td>253</td>
<td>23049</td>
<td>1784</td>
</tr>
</tbody>
</table>

Table II
PRIVACY LEAKS DETECTED IN BLACK MARKET APPLICATIONS

C. False Positives

Although every application that SCANDAL detected in-
cluded actual leaks, some of the paths were identified as
false positives. After computing a sound approximation of
every machine state of the Dalvik Core program, SCANDAL
reports a list of paths, each of which is represented as a pair
of an information source and a sink, which are potential
privacy leaks. We manually identified every path detected
from 11 Android Market applications. Table 3 summarizes the
result.

<table>
<thead>
<tr>
<th>Application</th>
<th>#path</th>
<th>#true</th>
<th>#false</th>
<th>#unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kids Preschool Puzzle</td>
<td>39</td>
<td>2</td>
<td>16</td>
<td>41</td>
</tr>
<tr>
<td>Job Search</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Kids Shapes</td>
<td>140</td>
<td>2</td>
<td>20</td>
<td>118</td>
</tr>
<tr>
<td>Kids ABC Phonics</td>
<td>59</td>
<td>2</td>
<td>16</td>
<td>41</td>
</tr>
<tr>
<td>Backgrounds HD Wallpapers</td>
<td>10</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Bible Quotes</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ES Task Manager</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Multi Touch Paint</td>
<td>80</td>
<td>2</td>
<td>6</td>
<td>72</td>
</tr>
<tr>
<td>Adobe File Manager</td>
<td>14</td>
<td>2</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>(D-Day) The Day Before</td>
<td>14</td>
<td>2</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Kids Numbers and Math</td>
<td>59</td>
<td>2</td>
<td>16</td>
<td>41</td>
</tr>
</tbody>
</table>

Table III
PATHS IDENTIFIED IN ANDROID MARKET APPLICATIONS

#path is the total number of reported paths. #true is
the number of actual leaks which are identified. #false is
the number of false positives which which are identified.
#unknown is the unidentified paths.

False positives mainly occur because the actual leaks
are exaggerated when analyzing library calls. SCANDAL
computes a sound approximation of privacy leak information
of an input application. When library functions are called
during a path of a privacy leak, SCANDAL soundly approx-
imates that private information can be exchanged between
its arguments in every possible way, which leads to an
exaggeration. Unless a library function call doesn’t flow
any information between its arguments and also discontinues
itself, which is rare, there exist at least one actual leak among
the exaggerated set of paths.

We failed to identify some paths for several reasons. Some
of third-party Android libraries included in the applications
were heavily obfuscated and made it hard to manually follow
the bytecode. Some of the applications heavily used java data
structures (array, list, map, ...) or JSON and made it hard to
accurately identify the data flow within their structure.

VI. LIMITATIONS AND FUTURE WORK

A. Performance

The time and memory consumption during the analysis
can be improved further. We have presented sparse [13]
and localization [12], [14] techniques. These are strong
and general optimization techniques for C static analyzer.
One of our main future work is to adopt such techniques
to SCANDAL and improve the performance. These per-
formance improvements will offer different scenarios and
possibilities for the use of the analyzer.

B. Java Native Interface

SCANDAL does not support the Java Native Interface
(JNI) methods. JNI defines a way to interact with native
code. Application developers may use JNI to incorporate
the C/C++ libraries into the application. Since the target
language of our analyzer is Dalvik VM bytecode, we are
unable to run our analysis on JNI libraries. 16 out of 90
Android Market applications we collected for the experiments
included JNI methods. Since it is possible to obtain private
information using JNI, it may cause new security problems.
Another one of our main future work is to extend the target
language of our analyzer to support JNI.

C. Reflection

SCANDAL does not fully support reflection-related APIs.
All 8 black market applications we collected for the ex-
periments used reflection while leaking private information.
We manually coded the semantics of reflection-related API
functions used in such cases. However, there are other ways
to use reflection, and we do not currently support them.
Also, SCANAL can only handle simple use of reflection. By using reflection, it is possible to instantiate new objects and invoke methods from their names. In all 8 applications, the names of the classes and methods were given as string constants. We believe that malicious developers can heavily obfuscate their behaviors using reflection. In such cases, our analyzer might not be able to precisely detect privacy leaks without adopting more complicated string analysis techniques.

D. Extending Analysis

Our static analysis framework can be extended to analyze other properties of Android applications. SCANAL is an abstract interpreter that computes a sound approximation of privacy leak information of an input application. By modifying domains and their abstractions, we can fairly easily achieve another abstract interpreter that verifies other semantical and interprocedural properties, such as array bound checking or dead code checking.

VII. RELATED WORK

Several tools and techniques have been presented to detect privacy leaks in smartphones using static analysis, but to our best knowledge, this is the first static analyzer based on the abstract interpretation framework for Android which targets Dalvik VM bytecode. PiOS [6] presented a static analysis for Objective-C code and detected privacy leaks in iPhone applications. DroidRanger [15] and Enck et al. [8] extensively studied Android market applications and gave better understanding of the current ecosystem. TaintDroid [7] and SCanDroid [11] are, respectively, dynamic and static analyzer detecting privacy leaks in Android applications.

TaintDroid [7] monitors Android applications at runtime, sacrificing runtime performance. TaintDroid monitors Android smartphone and tracks how applications leak private information. SCANAL is fully automated, while TaintDroid needs to execute an application to initiate privacy leaks. Also, TaintDroid causes about 14% CPU overhead for tracking, which is not an issue for SCANAL. We believe that by combining both approaches, both work can be complementary to each other. For example, if SCANAL guarantees an application to be safe, we can turn off the realtime monitoring while using the application.

SCanDroid [11] is limited because it cannot analyze packaged Android applications. SCanDroid is an automated static analyzer that reasons about data flows in Android applications. It checks whether data flows through an application are consistent with its permissions. However, it is not tested on real-world Android market applications. They used WALA, a collection of open-source libraries to analyze Java programs. So it cannot be immediately applied to packaged applications without the original Java code.

Enck et al. [8] used decompiling techniques that fail in some cases. They studied 1,100 popular free Android applications by using automated tools and manual inspections. They showed that Android applications often misuse users’ private data. They designed and implemented a Dalvik decompiler to recover Java source of an application before analyzing. However, they failed to recover the source code of about 5% of the total classes in the applications. SCANAL does not use reverse engineering techniques and deals directly with the bytecode. So we do not have to suffer from decompiling issues.

VIII. CONCLUSION

We provide a formal, sound, and automatic static analysis for detecting privacy leaks in Android applications. We tested our analyzer, SCANAL, with real-world applications, both from the official Android market and black markets. It detected privacy leaks in 11 applications out of 90 popular applications from Android market. It also detected various privacy leaks in 8 known malicious applications.

ACKNOWLEDGMENT

This work was supported by Samsung Electronics DMC R&D Center, the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea (NRF) (Grant 2012-000468), and the Brain Korea 21 Project, School of Electrical Engineering and Computer Science, Seoul National University.

REFERENCES


